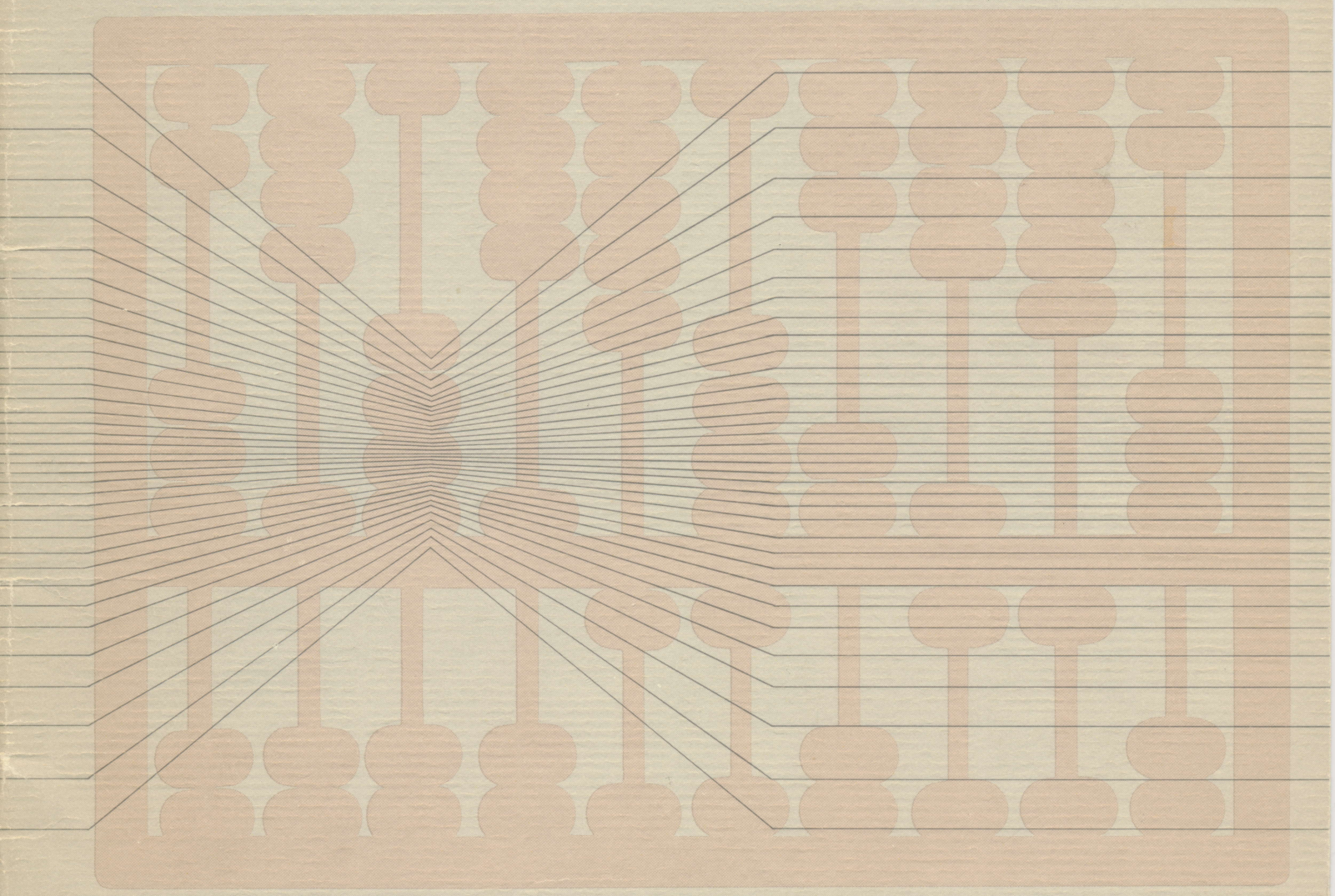
I.P. Sharp Associates

SHARP APL File System



SHARP APL
File System

Acknowledgements

The original development of the SHARP APL File System was a joint-development project between I.P. Sharp Associates Limited and Scientific Time-Sharing Corporation. Larry Breed, then of Scientific, and Eric Iverson, of I.P. Sharp Associates, jointly designed and implemented the system in 1969. Continuing support and development of the SHARP APL File System has been handled primarily by Peter Wooster.

The author of the first edition of this manual, in 1972, was Larry Breed. The manual was revised extensively by Brian Priebe and Leslie Goldsmith in 1974. Subsequent editions of the manual have profited greatly from the comments and criticisms of numerous readers.

Related Publications:

SHARP APL Reference Manual
File Conversions
SATN-8 Highspeed Print
SATN-10 SORTREQ

Revised April 1981.

Printed in Canada.
Publication Code: 210FA06
ISBN 0 86493 045 3

CONTENTS

		Page
	INTRODUCTION	1
1	FUNDAMENTALS OF FILE USE	2
2	ADDITIONAL PERSONAL FILE MANAGEMENT FACILITIES	7
3	FILE SHARING—CONCEPTS AND FUNCTIONS	11
4	DETAILED CONTROL OF FILE SHARING	17
	APPENDICES	
	A: Summary of File Functions	20
	B: Physical Storage Considerations	33
	C: Effect of System Failure on File Functions	34
	INDEX	35

CONTENTS

		Page
	INTRODUCTION	1
1	FUNDAMENTALS OF FILE USE	2
2	ADDITIONAL PERSONAL FILE MANAGEMENT FACILITIES	7
3	FILE SHARING--CONCEPTS AND FUNCTIONS	11
4	DETAILED CONTROL OF FILE SHARING	17
	APPENDICES	
	A: Summary of File Functions	20
	B: Physical Storage Considerations	33
	C: Effect of System Failure on File Functions	34
	INDEX	35

INTRODUCTION

The SHARP APL File System provides two fundamental capabilities:

1) The entry, retention, and access—under program control—of more data than can be stored in an APL workspace;

2) The sharing and selective modification of a data base by a group of users.

Chapter 1 of this manual covers fundamentals and the minimum set of file facilities needed for working with large amounts of data.

Chapter 2 explains additional file facilities useful in personal file management. Chapters 1 and 2 together include all the facilities needed to work with personal (non-shared) files.

Chapter 3 introduces file sharing. Facilities covered here allow both casual personal exchange of file information, and formal concurrent access to a common information base. Mechanisms are described for governing concurrency and detailed control of file access.

Chapter 4 discusses file security. It is intended for individuals who will be implementing major applications in inventory control, reservation, and management information systems, or other application areas requiring the use of shared files.

Appendix A explains all of the file functions.

Appendices B and C discuss some functional aspects of file use.

Chapter 1

FUNDAMENTALS OF FILE USE

A **file** is a collection of APL values, organized into **components**. A file is similar to a workspace in several respects:

It normally resides on an auxiliary storage device.

It may be created, used (and perhaps modified) over a period of time, and erased when it is no longer needed.

It is owned by a particular account number—generally, the account number of the person who created it.

There are file quotas. File quotas limit the amount of file storage that a user may occupy. Quotas are set by the APL Operator and can be increased or decreased by contacting the Operator or your SHARP APL representative.

It has a name and library number. Just as with workspaces, the library number may be either private (the owner's account number) or public (less than 1000).

A component of a file, like a variable in a workspace, may be any APL value—any scalar, vector, matrix, or higher dimensional array, holding either characters (including the character representation of a function) or numbers. It can also be a package. The only limit on the size of a component or variable is the implicit one that it should fit in a workspace.

There are also some important differences from workspaces:

More than one file may be active at the same time.

A file can hold large or small amounts of data. The size limit of a file may be specified by the owner and can range from a few thousand bytes up to hundreds of millions of bytes, or more. The size of a workspace is not variable.

A file is manipulated by system functions which can either be typed at the keyboard or executed within APL programs. A workspace is manipulated by system commands, most of which must be typed at the keyboard.

A file owner may specify **access control** for a file, to designate what other persons may access the file. Unless explicit authorization is given, the file is not available to anyone but the owner. A workspace is available to anyone who knows the workspace name, library number and password.

A variable in a workspace is referred to by its name. A component in a file is referred to by a **component number** that gives the component's position within the file. Component numbers are consecutive positive integers, ranging from the number of the first component up to the number of the last component.

File functions are used to bring components into the active workspace for processing, and to save values generated in the active workspace as components of a file. File functions are performed with SHARP

APL system functions. The names of these system functions indicate the kind of function being done. Examples of such functions are *⎕READ* (for reading a component from a file) and *⎕APPEND* (for appending to a file a value from the active workspace).

Improper use of file functions can lead to **file errors**. Such errors are indicated in the usual way by error reports. Appendix A summarizes each file function and the file errors it may engender. The syntax of the file functions shown in Chapters 1 through 3 omits the optional passnumber argument for simplicity, since it is used only in major shared-file applications. Appendix A gives the full syntax of the functions including the passnumber.

The library number and file name together are termed the **file id**. Where a file id is used as an argument to a file function, it is represented as a vector of characters within quotes (or as any other APL expression whose value is a character vector). The library number consists of 1 to 10 numeric characters. It may be omitted if it is the same as the sign-on number being used. The file name consists of 1 to 11 alphabetic and numeric characters, beginning with an alphabetic. If the library number is present, it is separated from the file name by one or more blanks. Leading and trailing blanks in the character vector are permitted.

File id's are patterned after workspace id's, but are logically distinct. There is no conflict in giving a workspace and a file the same name.

Before working with a file, an association must be made between the file id and a positive integer **file number**. All functions on the file will refer to the file by its number, rather than by its name. The particular value of the file number is unimportant as long as it is different from any other file numbers being used concurrently. The association between a file id and a file number is called a **file tie**, and a file together with its associated file number are said to be **tied**.

Descriptions and examples of file functions in Chapter 1 of this manual use the two files shown below. Entering the definitions of the programs used here will enable the reader to execute each example in turn.

The file named *PERSONS* has four components, each a vector of characters. The file *SALES* also has four components, each a vector of numbers.

component number	*PERSONS*	*SALES*
1	*'SMITH'*	5 6 3 1 4
2	*'JONES'*	2 6 10
3	*'KELLEY'*	4 6 2 9 1 5
4	*'BECKER'*	20 6 4

A new file is brought into existence with the ⎕*CREATE* function, and new components are placed on the file with the ⎕*APPEND* function. The following program may be used to build the two files, *PERSONS* and *SALES*.

```
    ∇ PROG1;PMAT;I;NAME
[1]   'PERSONS' ⎕CREATE 5
[2]   'SALES' ⎕CREATE 20
[3]   PMAT←4 6⍴'SMITH JONES KELLEYBECKER'
[4]   I←1
[5]  LOOP:NAME←PMAT[I;]
[6]   'ENTER VALUES FOR ',NAME
[7]   ⍞ ⎕APPEND 20
[8]   ((NAME≠' ')/NAME) ⎕APPEND 5
[9]   I←I+1
[10]  →LOOP×⍳I≤1↑⍴PMAT
[11]  ⎕UNTIE 20 5
    ∇
```

The ⎕*CREATE* function establishes a new file in a public library or in the user's private library, and makes it ready for further functions. The syntax is:

'file-id' ⎕*CREATE* file-number

The file id must be different from that of any existing file, and the file number must not be in use. The size limit on a newly created file is one hundred thousand bytes. When *PROG1* is executed, line 1 creates the file name *PERSONS* and ties it to file number 5. At this point, the file *PERSONS* exists in the file library, but it is empty since it contains no components so far. Components are added to the file *PERSONS* by the ⎕*APPEND* function on line 8.

The ⎕*APPEND* function puts a new component, containing an APL value from the active workspace, at the end of a file. The value in the workspace is not altered. The syntax is:

value ⎕*APPEND* file-number

The left argument which provides the value for the new component, may be the name of a variable or the result of any APL calculation. The right argument is the file number of the file being appended to (and, of course, may also be the result of an APL calculation).

Examples of appending to files *SALES* and *PERSONS* appear in *PROG1* on lines 7 and 8.

The ⎕*UNTIE* function has the syntax:

⎕*UNTIE* file-number-vector

It is used to break the association between a file and its file number. The argument is a vector of zero or more file numbers, so ⎕*UNTIE* can untie several files at once, as in line 11 of *PROG1*. ⎕*UNTIE* has no effect on the values stored in a file.

An existing file that is not tied can be made available for processing, as shown in the next program. This program prints each of the components of the file named *PERSONS*, along with the sum of the numbers in the corresponding component of *SALES*. The two file functions introduced in this example are ⎕*TIE* and ⎕*READ*.

```
    ∇ PROG2;X;I
[1]   'PERSONS' ⎕TIE 1
[2]   'SALES' ⎕TIE 2
[3]   'NAME       SUM'
[4]   I←1
[5] LOOP:X←⎕READ 2,I
[6]   (10↑⎕READ 1,I),⍕+/X
[7]   I←I+1
[8]   →LOOP×⍳I≤4
    ∇
```

The ⎕*TIE* function is used to establish an association between a file id and a file number. In *PROG2*, *PERSONS* is tied to file number 1, and *SALES* is tied to file number 2. The syntax of ⎕*TIE* is:

```
'file-id' ⎕TIE file-number
```

The ⎕*TIE* function has the same effect as ⎕*CREATE*, **except** that the file names in the left argument of ⎕*TIE* must already exist. Notice that in *PROG1*, *PERSONS* was tied to file number 5; in *PROG2*, *PERSONS* is tied to file number 1. File numbers may be chosen at will, and the number for a particular file can be different on different occasions.

The ⎕*READ* function copies the value of a file component into the active workspace. The value may be placed in a variable, as in line 5 of *PROG2*; it may be used in an expression, as in line 6; or it may be printed directly at the terminal. The syntax of ⎕*READ* is:

```
result ← ⎕READ file-number, component-number
```

The argument of ⎕*READ* is a two-element vector. The first element is the file number, and the second element is the component number. The symbols "result ←" in the syntax show that ⎕*READ* yields an explicit result and hence may appear within a larger expression. This is not true of functions such as ⎕*TIE* which have no explicit result.

Program *PROG2* tied the files *PERSONS* and *SALES*, but did not untie them, so those two files remain tied as files 1 and 2. A file tie is maintained until you untie it, erase it or sign off. The message *FILES UNTIED* will appear as part of the sign-off output if any files were still tied when you signed off. When you sign on after an involuntary sign-off (caused by a telephone interruption or local power failure), you must re-establish the appropriate ties in order to continue processing from the point of interruption. For example, suppose that while executing *PROG2* there had been an involuntary sign-off at line 5. When you sign on again, the *CONTINUE* workspace is automatically retrieved, and you may do the following to resume processing:

```
      )SI
PROG2[5]   *
      'PERSONS' ⎕TIE 1
      'SALES' ⎕TIE 2
      →5
```

The status of tied files is not changed by any system commands except)*OFF* and)*CONTINUE*. Thus, loading into, or clearing the active workspace, leaves active any ties established previously during the terminal session. This makes it possible to load different workspaces to process files without having to re-establish the ties.

The names of the files in a particular private or public library are obtained with the □*LIB* function. The syntax is:

result ← □*LIB* library-number

The result is a character matrix. Each row holds the identification of a file, in the designated library, that the person executing □*LIB* is authorized to access. (When applied to someone else's private library, □*LIB* will return an empty matrix unless that person has explicitly given you permission to access some of his/her files.) If, for instance, your account number is 1234, the expression:

```
□LIB 1234
```

produces this result:

```
1234 PERSONS
1234 SALES
```

Each row of the matrix is a file id. The number of files in the library is equal to `1↑⍴□LIB 1234`.

When a file is no longer needed, it can be erased:

'file-id' □*ERASE* file-number

The file name is deleted from the library, all components are destroyed, the space they occupied is made available for use by other files, and the file tie is broken. Note that in order to erase a file it must be tied, and that both the file name **and** the file number must be given.

The following program is an example of changing the arrangement of some filed information. The components of *PERSONS* and *SALES* are to be merged into a new file, *RECORDS*. Each odd-numbered component of *RECORDS* will come from *SALES*. The two original files are erased after the function is finished. Recall that *PERSONS* and *SALES* are still tied following execution of *PROG2*.

```
    ∇ PROG3;I
[1]   'RECORDS' □CREATE 3
[2]   I←1
[3]  LOOP:(□READ(1+2|I-1),⌈I÷2) □APPEND 3
[4]   →LOOP×⍳8≥I←I+1
[5]   □UNTIE 3
[6]   'SALES' □ERASE 2
[7]   'PERSONS' □ERASE 1
    ∇
```

Line 3 shows how an APL expression can be used to calculate the value of a file number or a component number. In this example, the file numbers 1 and 2 did not appear in a □*UNTIE* because erasing *PERSONS* and *SALES* also unties them.

Chapter 2

ADDITIONAL PERSONAL FILE MANAGEMENT FACILITIES

This chapter covers the remaining file functions that are used with non-shared files.

Two niladic functions, ⎕*NAMES* and ⎕*NUMS*, report on the status of file ties. ⎕*NAMES* has the syntax:

```
result ← ⎕NAMES
```

and returns a matrix of characters, in which each row is the library number and name of a file currently tied. ⎕*NUMS* returns a vector of the file tie numbers corresponding to the names given by ⎕*NAMES*. The syntax of ⎕*NUMS* is:

```
result ← ⎕NUMS
```

The simplest way to untie all tied files is by executing the statement ⎕*UNTIE* ⎕*NUMS*, since the vector returned by ⎕*NUMS* is exactly what ⎕*UNTIE* needs in order to untie all files.

A file component may be replaced by a new APL value, using the ⎕*REPLACE* function. The syntax is:

```
value ⎕REPLACE file-number, component-number
```

For example, to replace the second component of file number 1 by the reciprocals of the first 50 integers, this statement could be used:

```
(÷⍳50) ⎕REPLACE 1 2
```

The new value of the component may be any APL value that fits in the active workspace. Any additional file storage that is needed to hold the new value is provided automatically.

A common use of ⎕*REPLACE* is to update a component by modifying its value. In this example, the value of the variable *DEBIT* is catenated to the existing value of component 3 of file number 7.

```
((⎕READ 7 3),DEBIT) ⎕REPLACE 7 3
```

File storage requirements can increase markedly if many small components are replaced by larger components, as in this example. For a more detailed discussion of file storage, see Appendix B—Physical Storage Considerations.

The ⎕*DROP* function removes components from either end of a file. The syntax is:

```
⎕DROP file-number, n
```

If n is positive, n components are removed from the low-numbered end of the file; if n is negative, |n components are removed from the high-numbered end of the file. If n is zero, no components are removed from either end of the file. The component numbers of the remaining components are not changed; for example, if file 99 has ten components numbered 1 through 10, then after executing:

```
□DROP 99 4
```

the file will have six components remaining, numbered 5, 6, 7, 8, 9 and 10. A further execution of:

```
□DROP 99 ¯2
```

will leave components 5, 6, 7 and 8.

Components may not be dropped from the interior of a file. Other techniques may be used to signify that an interior component holds no information and should be bypassed in later processing; perhaps the simplest is to replace the component by an empty vector, ⍬.

The function □*SIZE*, with syntax:

```
result ← □SIZE file-number
```

returns a four-element vector. The first element is the number of the first component of the file. The second element is the number of the last component **plus** one. The third and fourth are the amount of file storage currently occupied (the **file size**) and the file size limit, in bytes.

Here is how two common file organizations, first-in-first-out (FIFO) and last-in-first-out (LIFO), may be implemented using □*DROP* and □*SIZE*.

FIFO organization:

Data collection

```
DATA □APPEND 50
```

Processing

```
INFO ← □READ 50,(□SIZE 50)[1]
(process INFO)
□DROP 50 1
```

LIFO organization:

Data collection

```
DATA □APPEND 50
```

Processing

```
INFO ← □READ 50,(□SIZE 50)[2]-1
(process INFO)
□DROP 50 ¯1
```

With each of these schemes, the data collection and the processing phases may be executed when it is convenient to do so—the data collection phase whenever information is available and the processing phase when the file grows too large or at some fixed interval (e.g., once a day or once a week). After a component is processed, the storage it occupies is released by the □*DROP*, so the file can be built and processed almost indefinitely and never require more than a relatively small amount of storage space.

As an example of using ⎕*DROP* with a file whose components are released in an arbitrary sequence, the following program produces an invoice from information in a file named *CDATA*. After the invoice is produced, the appropriate component of *CDATA* is dropped (if it is the first component) or replaced with an empty vector (if it is not the first component). Each time the program is used, it checks to see if the first component of *CDATA* is an empty vector. If it is, then it is dropped. This technique will tend to keep the file size to a minimum.

```
    ∇ PROG4;N;RECORD;LIMITS
[1]   'CDATA' ⎕TIE 5
[2]   'ENTER RECORD NUMBER'
[3]   N←⎕
[4]   →ERROR×⍳N=.<2↑⎕SIZE 5
[5]   RECORD←⎕READ 5,N
[6]   →ERROR×⍳0=⍴RECORD
[7]   PRINTINVOICE
[8]   (⍳0) ⎕REPLACE 5,N
[9]  LOOP:→OUT×⍳=/LIMITS←2↑⎕SIZE 5
[10]  →OUT×⍳0≠⍴⎕READ 5,1↑LIMITS
[11]  ⎕DROP 5 1
[12]  →LOOP
[13] ERROR:'THIS NUMBER NOT IN FILE'
[14] OUT:⎕UNTIE 5
    ∇
```

In the program *PROG4*, line 4 causes a branch to *ERROR* if *N* is not within the range of the lowest and highest component numbers. Otherwise, line 5 reads the component. If the component is empty, a branch is taken to *ERROR*. Line 7 actually prints the invoice using the function *PRINTINVOICE* (not shown). Lines 9 through 12 drop empty components from the file. Note the checking for an empty file on line 9.

The ⎕*RENAME* function is used to change a file id. The syntax is:

```
'file-id' ⎕RENAME file-number
```

⎕*RENAME* can be used to change the name of a file in a personal library:

```
'REPORTS' ⎕TIE 100
'OLDREPORTS' ⎕RENAME 100
⎕UNTIE 100
```

or to move a file from another person's library into your own:

```
'5440 FIGHT' ⎕TIE 1844
'OREGON' ⎕RENAME 1844
⎕UNTIE 1844
```

This can be done only if the owner of the file has given you permission to rename the file.

Note that ⎕*RENAME* does **not** create a second copy of a file—after the execution of the previous example, *FIGHT* is no longer in library 5440.

The function ⎕*RESIZE* is used to change the size limit of the file. The syntax is:

```
size-limit ⎕RESIZE file-number
```

The size limit for *PICAYUNE* could have been changed to about four million bytes in the following manner:

```
      4000000 ⎕RESIZE 30
```

The function ⎕*AVAIL* is used to determine whether or not the file system is operating. The syntax is:

```
result ← ⎕AVAIL ⍳0
```

⎕*AVAIL* requires a dummy right argument of ⍳0 to allow for future extension.

If the file system is functional, the result of ⎕*AVAIL* will be a 1; otherwise, it will return a 0. ⎕*AVAIL* will normally return 0 when APL service is just starting or ending. If the file system is not operating at any other time, the APL Operator should be consulted.

The function ⎕*APPENDR* puts a new component, containing a value from the active workspace, at the end of the file **and** returns the number of the component. The syntax is:

```
result ← value ⎕APPEND file-number
```

Chapter 3

FILE SHARING—CONCEPTS AND FUNCTIONS

The shared use of a file falls into one of two general categories. In the first, only one task at a time may use a file (that is, have it tied). This resembles several people making use of a single typewriter. The second category of sharing wherein several tasks use the file simultaneously resembles the concurrent use of several telephone receivers on one telephone number, which may require some protocol among speakers to prevent conflicts.

Associated with every file is an **access matrix** that records which account numbers are authorized to use the file, and which file functions each account number is allowed to access it by. An access matrix is integer-valued, with three columns and any number of rows. Each row represents authorization for a single account number. The first element of the row is the account number; the second is an encoding of permitted file functions; and the third is a **passnumber**.

The account number may be that of any individual, or zero. Zero represents all account numbers.

When a file is tied, the access matrix for the file is searched for a row describing the user's account number and passnumber. If a row is found whose first **and** third elements match the account number and passnumber respectively, or—failing that—if a row is found whose first element is zero **and** whose third element matches the passnumber, then the authorized forms of access are taken from column two of that row. Any attempt to use a file function that isn't authorized will result in a *FILE ACCESS ERROR* report, and a notation of the access error will also be placed in a special system file that is examined periodically by the computer operations staff. If the value for □*TIE* is not included in column two, the □*TIE* itself will fail with a *FILE ACCESS ERROR* report.

The owner of a file has complete access to the file with a passnumber of 0 unless some row of the access matrix contains the owner's account number, with a passnumber of 0. In such a case, the owner's access is limited by the file functions designated. No one else has access unless there is a match with their account number or a zero in column one; they will receive a *FILE ACCESS ERROR* report if they attempt to tie the file.

Functions like □*NAMES* and □*NUMS* are not restricted by access matrix settings and never produce *FILE ACCESS ERROR* reports, because they are not functions that are performed upon a particular file. Also, there is no restriction on □*UNTIE* or □*STIE*; these functions are permitted if **any** other function is authorized.

Recall that the □*LIB* function returns the file id's of all files in a library to which the person executing the □*LIB* is permitted some form of access. Thus, for instance, □*LIB* `100` will give different results when executed by different people if there are files in library 100 to which not everyone has access.

The functions □*STAC* and □*RDAC* are used to manipulate the access matrix. The value of the access matrix is set using the □*STAC* function. The syntax is:

```
value □STAC file-number
```

The left argument must be an integer-valued, three-column matrix. It replaces the previous value of the access matrix. Initially, when a file is created, its access matrix has no rows—its shape is 0 3. The file owner has complete access with passnumber 0, and no one else has any access.

The access matrix is read into the active workspace using the ⎕*RDAC* function. The syntax is:

result ← ⎕*RDAC* file-number

The result is the access matrix for the file designated by the right argument.

The second element in a row of the access matrix is the sum of coded values for the particular functions that are allowed. The values are:

1	⎕*READ* and ⎕*SIZE*
2	⎕*TIE*
4	⎕*ERASE*
8	⎕*APPEND*
16	⎕*REPLACE*
32	⎕*DROP*
64	⎕*HOLD*
128	⎕*RENAME*
256	⎕*RDAC* and ⎕*STAC*
512	⎕*RDCI*
1024	⎕*RESIZE*
2048	⎕*FHOLD*
4096	⎕*RDAC* only
8192	⎕*STAC* only
16384	⎕*APPENDR*

Thus a value of 10 indicates authorization to execute ⎕*TIE* and ⎕*APPEND* functions on this file. A value of ¯1 indicates authorization for all functions. Any non-zero value permits use of ⎕*STIE* and ⎕*UNTIE*.

Negative numbers can be conveniently used to deny certain accesses (i.e., "permit everything but"). In general ¯1-n (or -1+n) will restrict accesses specified by n. For example ¯5 will permit all but ⎕*ERASE* (although it does permit ⎕*STAC*, so the file can be erased with a 2-step procedure).

The third element in a row of the access matrix, the **passnumber**, is usually zero. Non-zero passnumbers are used only to exercise detailed control over file access and will be discussed in Chapter 4.

Along with the value stored in a component by appending or replacing, three other pieces of information are carried with each component of a file: the workspace storage needed to hold the component's value; the account number which last replaced or appended the component; and the time that the component was last replaced or appended. The ⎕*RDCI* function reads this "component information". The syntax is:

result ← ⎕*RDCI* file-number, component-number

The result is a three-element vector holding the described information. The time, in the third element of the result, represents 60ths of seconds since midnight, March 1, 1960. Workspace 1 *FILEAID* contains two functions for representing the time as given by ⎕*RDCI* in a choice of formats. The function *TIMEN* returns a seven-element numeric representation consisting of year, month, day, hour, minute, second, and 60ths of seconds. The function *TIME* returns character representations of the form *MM/DD/YY HH:MM:SS:XX* (where *XX* represents sixtieths of a second).

The ⎕*RDCI* function is particularly useful in data collection or audit trail applications, where several accounts may have access to append to a file. As each appends, the component is automatically tagged

with the time and the account number. Later, when an account number with ⎕*RDCI* authorization processes the file, the source of each component is clearly indicated.

The foregoing discussion covers the first category of shared file use, in which only one task at a time is using a file. The use of ⎕*TIE* to establish a file tie assures this kind of file sharing, since ⎕*TIE* will fail with a *FILE TIED* report if any other task already has the file tied. Thus ⎕*TIE* provides one task with exclusive use of a file until it is untied.

The second category of file sharing covers situations in which several tasks may concurrently have the same file tied. Two new functions are needed for concurrent file use: ⎕*STIE* and ⎕*FHOLD*.

'file-id' ⎕*STIE* file-number

establishes a tie to a file, exactly as ⎕*TIE* does. However, this "shared tie" permits other tasks to establish shared ties to the same file. A ⎕*STIE* may be executed if the access matrix for the file allows any form of access whatever (as long as no one else has the file ⎕*TIE*'d).

When several tasks are using a file concurrently, their file functions proceed asynchronously and it is quite likely that one task's processing may be interleaved with another's. Suppose, for example, that tasks *P* and *Q* are each trying to add 1 to the value of component 5, file 77, by executing a program containing the statement:

```
(1+⎕READ 77 5) ⎕REPLACE 77 5
```

When both have finished, component 5 should be increased by 2. However, suppose the parts of the foregoing statement are executed in this sequence:

P	*Q*
	⎕*READ* 77 5
⎕*READ* 77 5	
	(add 1)
	⎕*REPLACE* 77 5
(add 1)	
⎕*REPLACE* 77 5	

The value of component 5 is increased by only 1, since *P* destroyed the value stored by *Q*.

What is needed here is an interlock to prevent *P* from executing any part of the foregoing APL statement while *Q* is executing it (and vice versa). The ⎕*FHOLD* function provides this interlock. The syntax is:

⎕*FHOLD* vector of file-numbers

The effect of executing ⎕*FHOLD* is to place an interlock on each file mentioned in the right argument. A ⎕*FHOLD* executed by *P* operates in three stages:

1) Any interlocks in effect from a previous ⎕*FHOLD* executed by *P* are released.

2) *P* is placed in a queue behind every other task that is *FHOLD*ing (or waiting to ⎕*FHOLD*) any of the files mentioned in the right argument. *P* is delayed until no one else is ⎕*FHOLD*ing any of those files. (There is no delay if all the files are available—in other words, if the queue is empty.)

3) Interlocks are set simultaneously on all the designated files, and *P* resumes execution.

The use of □*FHOLD* is almost always simpler than this procedure sounds. Note that □*FHOLD* ⍳0 has the effect of releasing all interlocks (stage 1) and proceeding without delay since stages 2 and 3 have no effect. The interlocks set by a □*FHOLD* are released by any of the following:

1) Executing another □*FHOLD*;
2) Returning to immediate-execution (desk calculator) mode—for instance, following an execution error or an attention signal;
3) Signing off, either voluntarily or involuntarily.

In particular, executing □*FHOLD* in immediate-execution mode rather than from within a program has no effect since (by point 2) any interlocks are immediately released before the next keyboard entry is accepted.

Now suppose the program being executed by *P* and *Q* in the preceding example is changed to this:

```
□FHOLD 77
(1+□READ 77 5) □REPLACE 77 5
□FHOLD ⍳0
```

Then, the first task to execute the □*FHOLD* 77 is able to proceed, while the other task is delayed **until** the first executes □*FHOLD* ⍳0. The sequence will look something like this:

```
    P                   Q
                        □FHOLD 77
                        (proceed)
                        □READ 77 5
□FHOLD 77
(delay)
                        (add 1)
                        □REPLACE 77 5
                        □FHOLD ⍳0
(proceed)               (proceed)
□REPLACE 77 5
□FHOLD ⍳0
(proceed)
```

It is now impossible for more than one task at a time to be executing the statement between the □*FHOLD*s, and so component 5 of the file 77 is assured of being increased correctly.

It is important to remember that the **only** effect of an interlock set by a □*FHOLD* is to delay the execution of other users' □*FHOLD*s; it does not affect the execution of any other file function such as □*READ* or □*REPLACE*. Thus, if user S were to execute □*READ* functions on file 77 in the example above, the □*READ*s would be completely unaffected by the presence or absence of □*FHOLD* interlocks set by *P* and *Q*.

A program for terminal-to-terminal conversation between two people with each "talking" in turn, may be implemented as follows. (The person with account number 1001 creates a file and appends one component for an initial message.)

```
      'TALKTALK' ⎕CREATE 1
      'HELLO' ⎕APPEND 1
```

Account number 1002 is given permission to read, hold, and replace in file 1001 *TALKTALK*. Then, from two different terminals, 1001 and 1002 each execute this function:

```
    ∇ CONVERSE;T
[1]   '1001 TALKTALK' ⎕STIE 1
[2]  MORE:⎕FHOLD 1
[3]   ⎕READ 1 1
[4]   →OUT×⍳0=⍴T←,⍞
[5]   T ⎕REPLACE 1 1
[6]   →MORE
[7]  OUT:'GOODBYE' ⎕REPLACE 1 1
[8]   ⎕UNTIE 1
    ∇
```

Now each person in turn receives a message (printed on the terminal by line 3) and sends a message (line 5). The queuing in stage 2 of ⎕*FHOLD* execution assures that the two take turns in order. Multilateral games, exercises in strategic decision-making, and simulations of asynchronous but interdependent activities (for example, multiprocessing computers) can be implemented by employing variants of this technique.

The use of ⎕*FHOLD* to control sequencing between programs is not always associated with the use of ⎕*REPLACE*. Here are some situations that do or do not require ⎕*FHOLD* for proper function:

1) Several users are concurrently appending to a file but making no other use of the file. ⎕*FHOLD* is not needed. Although it is not possible to predict in what order the various values will appear in the file, any request for a file function (in this case ⎕*APPEND*) always waits until a previous function on the same file is complete.

2) Several users are concurrently appending, as in situation 1, but each is using ⎕*SIZE* to determine the component number of their component. This pair of statements is being used:

```
MYCNO ←(⎕SIZE 100)[2]
MYVAL ⎕APPEND 100
```

   A ⎕*FHOLD* 100 is required **before**, and a ⎕*FHOLD* ⍳0 **after** these statements. Without the ⎕*FHOLD*s there is no assurance that *Q* will not do a ⎕*APPEND* after *P* has executed the ⎕*SIZE* but before *P* has executed the ⎕*APPEND*. If this were to happen, the value of *MYCNO* in *P*'s program would be incorrect.

   This situation can be solved in a better way, however. The function ⎕*APPENDR* can be used to perform the append and capture the resulting component number in a single operation.

   The example could be coded as:

```
MYCNO←MYVAL ⎕APPENDR 100
```

3) Several users are reading and replacing components of a file, and it is known that no two users ever reference the same component. (For instance, P's program refers only to component 1, Q's program only to component 2, and so on.) No □*FHOLD* is needed since no conflict can occur on the concurrent use of a single component.

4) An application involves the use of three files in which like-numbered components contain related data. One program is updating the files, while others are reading the files concurrently. To ensure that no program that is reading will encounter a mixture of old and new data, the updating program has this appearance:

```
□FHOLD 21 22 23
(process and replace component I of each of the three files)
□FHOLD ⍳0
```

and the file-reading programs that are operating concurrently have this appearance:

```
□FHOLD 21 22 23
A ← □READ 21,J
B ← □READ 22,J
C ← □READ 23,J
□FHOLD ⍳0
(process data in A, B, and C)
```

This is a situation in which □*FHOLD* is needed in a program that is itself not modifying the contents of any files.

As these examples show, the need for □*FHOLD*s is dependent on the interrelation of program and file structure. The designer of any application involving concurrent use of files must analyze it carefully for possible "races" between asynchronous programs, and resolve them through appropriate use of □*FHOLD*.

Chapter 4

DETAILED CONTROL OF FILE SHARING

This chapter covers some mechanisms that can be used for limiting file access to particular parts of a file, or to specific sequences of file functions. These mechanisms are of interest primarily to designers of large-scale shared-file applications; most readers will find the facilities described in Chapter 3 adequate for their needs.

Detailed control over file access may be achieved by requiring a non-zero passnumber. Users must tie the file using the passnumber which appears with their sign-on number in the access matrix. Furthermore, most other file functions acting on the file must provide that passnumber. Appendix A describes which functions require passnumbers.

Omitting a passnumber from an argument (as in all uses of file functions in Chapters 1-3) is equivalent to providing an explicit passnumber of zero. A mismatching passnumber will cause a *FILE ACCESS ERROR* report which is automatically noted in a special file for the computer operations staff.

Passnumbers are intended for use within locked APL functions that the application designer makes available to his users. For example, suppose each component of the personnel file 2000 *PERS* is a vector, holding an employee's telephone extension and room number followed by salary information. Account number 1234 is to be allowed to interrogate the telephone and room number, but not the salary information.

The designer chooses a passnumber—say, 161803—and defines these locked functions:

```
    ∇ PTIE
[1]   '2000 PERS' ⎕STIE 25 161803
    ∇

    ∇ R←PREAD N
[1]   R←2↑⎕READ 25,(N),161803
    ∇
```

Next, the designer sets the access matrix for 2000 *PERS* to authorize access by account number 1234. The row of the access matrix will be:

```
1234  1  161803
```

Finally, the designer gives user 1234 the locked functions, but **does not reveal** the passnumber. Now user 1234 can read the file (1 designating ⎕*READ* permission), but **only** if the passnumber, 161803, is given with each ⎕*READ*. Since the passnumber is not known, the user can only access file 2000 *PERS* through the functions *PTIE* and *PREAD*.

Using this general technique, it is possible to impose quite complex restrictions on file authorization—in fact, any sort of restriction that can be stated as an APL program. Since different passnumbers may be imposed on different account numbers, it is easy to provide multiple levels of access authorization to confidential data. The previous example showed access restricted to specific fields within a component. Examples of other forms of control are:

access only to even-numbered components:

```
[1]   →ERR×⍳0≠2|N
[2]   R←⎕READ 25,(N),161803
```

access only at certain times of day:

```
[1]   →ERR×⍳~⎕TS[4]∈11 12 13 14
[2]   R←⎕READ 25,(N),161803
```

automatic logging of information requests:

```
[1]   N ⎕APPEND 99 316227
[2]   R←⎕READ 25,(N),161803
```

(later use of ⎕*READ* and ⎕*RDCI* on file 99 will give the value of *N* and the timestamp and requestor's account number.)

access only after validating data:

```
[1]   →ERR×⍳1≠⍴⍴V
[2]   →ERR×⍳(20≠⍴V)∨0∨.>V
[3]   V ⎕APPEND 45 141421
```

access for a particular sequence of functions:

```
[1]   ⎕FHOLD 30
[2]   (1+⎕READ 30 1 5772157) ⎕REPLACE 30 1 5772157
[3]   ⎕FHOLD ⍳0
```

access only after verifying user identity by questions and answers to protect a momentarily unattended terminal from passersby;

access only to summary data (for instance, salary averages but no individual salaries);

and even access for clerks to authorize access for other users without being able to affect their own or their superiors' access authorization.

Preparation of a package of functions for manipulation of confidential information requires observance of some precautions, to assure integrity of file data and of the programs themselves. Such a package must prevent unauthorized display or modification of file components, passnumbers, or programs. Attempts to violate file security can be averted by observing these guidelines:

1) Write passnumbers as constants—never define a variable with the value of the passnumber and then use the variable. Avoid defining variables containing data values that the user is not authorized to display. Program execution might stop within a function due to an attention signal or a program error, and with execution suspended, examination of local variables could disclose passnumbers or data values. For instance instead of:

```
DIRECTORY←⎕READ 10 1 311037
R←DIRECTORY[;1]⍳1↑⎕AI
```

write:

```
R←(⎕READ 10 1 311037)[;1]⍳1↑⎕AI
```

2) When designing ⎕*FHOLD* interlocks or other means of communicating between users, take into consideration the effects of possible telephone interruptions during program execution. Where appropriate, use the timestamp returned by ⎕*RDCI* to check whether a telephone interruption or attention signal has allowed some other user to proceed with file modification. Or a check could be made on keying time (`⎕AI[4]`).

3) If a file component may be modified by several users, try to use a single APL statement to read, modify and replace the component so that program execution cannot be interrupted and later resumed within a critical area. This can be done quite easily by employing the ◇ (diamond) symbol. For example, instead of:

```
[1]   A←⎕READ 10 1 311037
[2]   A[;1]←DATA
[3]   A ⎕REPLACE 10 1 311037
```

write

```
[1]   A←⎕READ 10 1 311037 ◇ A[;1]←DATA ◇ A ⎕REPLACE 10 1 311037
```

4) In several examples above, some seemingly superfluous parentheses have been employed. For example:

```
R←⎕READ 25,(N),161803
```

This trick ensures that the passnumber cannot be passed to the function *N* as a right argument. Of course *N* is supposed to be a variable, but a malicious user might interrupt the function and modify *N* to be a monadic function which displays its right argument. If the line had been coded:

```
R←⎕READ 25,N,161803
```

and the user branched to this line, the passnumber would be discovered.

Appendix A

SUMMARY OF FILE FUNCTIONS

Function	Syntax	Permission
□*APPEND*	expr □*APPEND* fn,pn	8
□*APPENDR*	r←expr □*APPENDR* fn,pn	16384
□*AVAIL*	r←□*AVAIL* ⍳0	
□*CREATE*	fnm □*CREATE* fn	
□*DROP*	□*DROP* fn,n,pn	32
□*ERASE*	fnm □*ERASE* fn,pn	4
□*FHOLD*	□*FHOLD* svm	2048
□*HOLD*	□*HOLD* fn(s)	64
□*LIB*	r←□*LIB* libno	
□*NAMES*	r←□*NAMES*	
□*NUMS*	r←□*NUMS*	
□*RDAC*	r←□*RDAC* fn,pn	4096,256
□*RDCI*	r←□*RDCI* fn,cn,pn	512
□*READ*	r←□*READ* fn,cn,pn	1
□*RENAME*	fnm □*RENAME* fn,pn	128
□*REPLACE*	expr □*REPLACE* fn,cn,pn	16
□*RESIZE*	p □*RESIZE* fn,pn	1024
□*SIZE*	r←□*SIZE* fn,pn	1
□*STAC*	mat □*STAC* fn,pn	8192,256
□*STIE*	fnm □*STIE* fn,pn	
□*TIE*	fnm □*TIE* fn,pn	2
□*UNTIE*	□*UNTIE* fn(s)	

cn	Component number
expr	Data
fn	File number
fnm	File name
libno	Library number
mat	3-column access matrix
n	Integer
p	Positive integer
pn	Passnumber or ⍳0
r	Result
svm	Scalar, vector, or 2-row matrix of integers

The syntax for fnm of □*CREATE* and □*RENAME* is '[library-number] filename [file-size-limit]'. Parameters in square brackets [ ] are optional.

Complete access to a file can be given by a permission code of ¯1.

This appendix gives the syntax of each file function, a description of its effect, and possible error conditions. The syntax lines in the descriptions of file functions use the symbols shown below. The actual arguments of file functions can of course arise from evaluation of any APL expressions. The symbols used in a syntax line simply show how various portions of the arguments will be interpreted.

component-number	An integer designating the position of a component in a file.
file-id	A character vector containing a file name optionally preceded by a library number.
file-number	An integer designating a file tie number.
file-number-passno-matrix	A scalar or vector of file-numbers **or** a two-row matrix of file numbers and corresponding passnumbers.
file-number-vector	A vector of 0 or more file numbers.
int	An integer designating the number and position of components to be dropped.
library-number	An integer designating a public or private library.
passno	An integer passnumber.
size-limit	A character vector of decimal digits giving maximum file size in bytes.
result	The result, if any, of a file function.
value	Any APL scalar, array, or package.
←	Indicates that the file function returns a result.
`[ ]`	Encloses optional quantities that may or may not appear in a particular use of a file function. For instance, `m[,n]` indicates that both m and m,n are permitted.
`' '`	Indicates character-valued information.

The file function descriptions in this section give the syntax of the file function and only those error reports that are produced by that particular file function. Other errors, which do not depend on the particular file function, are shown here.

DOMAIN ERROR	An argument of the file function is not integer-valued. e.g., `⎕READ 1 1.5` A file-id is not properly formed. e.g., `'AG∈' ⎕STIE 1`
FILE BACKUP INTERRUPT	You have signalled an interrupt (2 successive uses of ATTN or BREAK) and have interrupted a file operation which is delayed during a system file backup. Try again in a few minutes.
FILE INTERRUPT	You have signalled an interrupt (2 successive uses of ATTN or BREAK) and have interrupted a file operation. Retry the operation.
FILE SYSTEM NOT AVAILABLE	The file system is not available for use. Consult the APL Operator for assistance.
LENGTH ERROR	An argument to a file function has an incorrect number of elements. e.g., `⎕READ 2 3 4 5 6 7`
RANK ERROR	The rank of an argument to a file function is incorrect. e.g., `⎕READ 1 2⍴1 1`
SYNTAX ERROR	The file function has been used dyadically instead of monadically or vice versa. e.g., `⎕APPEND 1`
WS FULL	Available workspace storage is insufficient to invoke the file function and return the concomitant result.

Several error reports are possible which report difficulties with the file system itself. These are extremely rare but are included here for completeness. In every case, contact the APL Operator or your SHARP APL representative to correct the problem.

FILE COMPONENT DAMAGED	A system crash has resulted in a damaged file component.
FILE DAMAGED	A system crash has resulted in a damaged file.
FILE LIBRARY NOT AVAILABLE	The volume containing the file you have requested is unavailable at present. This is usually because of maintenance or trouble with the physical device on which that volume is stored.
FILE SYSTEM ERROR	There is something seriously wrong with the file system.
FILE SYSTEM HARDWARE ERROR	A hardware error is preventing the file system from accessing your file.
FILE SYSTEM NO SPACE	All SHARP APL file system storage is consumed; no more data can be saved; a temporary condition.

FILE SYSTEM TIES USED UP Total file system usage is so high that no more files can be tied; a temporary condition.

A file space quota, a file quota, and a file tie quota, are imposed to limit the system resources allocated to a single user. The amount of file space that may be reserved by an account number is set, and may be changed by the APL Operator. It limits the sum of the file size limits (the fourth element of *□SIZE*) for all files owned by that account number. The error report:

FILE RESERVATION ERROR

on executing *□CREATE*, *□RENAME* or *□RESIZE*, indicates an attempt to exceed the account number's file space quota.

The total number of files that may be owned by an account number is limited by a file quota. The error report:

FILE QUOTA USED UP

on executing *□CREATE* or *□RENAME* indicates an attempt to exceed this quota. As with the file space quota, to continue, either decrease the resource consumed (e.g., *□ERASE* some files) or contact the APL Operator or your SHARP APL representative to have your quota(s) increased.

The number of file ties that may be concurrently established by a single task is limited to twenty. This is a system-wide quota that applies to all account numbers. The report:

FILE TIE QUOTA USED UP

on executing *□CREATE*, *□TIE*, or *□STIE*, indicates that twenty files are already tied.

The table below summarizes the information presented on the pages which follow. For each error message, the X's indicate those functions that can produce that message.

	File Access Error	File Full	File Index Error	File Name Error	File Quota Used Up	File Reservation Error	File Tie Error	File Tie Quota Used Up	File Tied
⎕*APPEND*	X	X	X				X		
⎕*APPENDR*	X	X	X				X		
⎕*AVAIL*									
⎕*CREATE*	X			X	X	X	X	X	
⎕*DROP*	X		X				X		
⎕*ERASE*	X			X			X		X
⎕*FHOLD*	X						X		
⎕*HOLD*	X						X		
⎕*LIB*									
⎕*NAMES*									
⎕*NUMS*									
⎕*RDAC*	X						X		
⎕*RDCI*	X		X				X		
⎕*READ*	X		X				X		
⎕*RENAME*	X	X		X	X	X	X		
⎕*REPLACE*	X	X	X				X		
⎕*RESIZE*	X	X				X	X		
⎕*SIZE*	X						X		
⎕*STAC*	X						X		
⎕*STIE*	X			X			X	X	X
⎕*TIE*	X			X			X	X	X
⎕*UNTIE*							X		

value □*APPEND* file-number [,passno]

Creates a new component with value equal to the left argument and component number one greater than the greatest previous component number.

FILE ACCESS ERROR	- User not permitted to append to this file; or incorrect pass-number.
FILE FULL	- The new component was not appended because the file already consumed or exceeded its storage reservation.
FILE INDEX ERROR	- The component number would exceed 2147483647.
FILE TIE ERROR	- File number not tied to a file.

result ← value □*APPENDR* file-number [,passno]

Creates a new component with value equal to the left argument and component number one greater than the greatest previous component number. Result is the component number of the appended value.

FILE ACCESS ERROR	- User not permitted to append to this file; or incorrect pass-number.
FILE FULL	- The new component was not appended because the file already consumed or exceeded its storage reservation.
FILE INDEX ERROR	- The component number would exceed 2147483647.
FILE TIE ERROR	- File number not tied to a file.

result ← □*AVAIL* ⍳0

Returns a scalar 1 or 0. 1 indicates that the file system is operating; otherwise, a 0 will be returned. A result of 0 is normal when APL service is just starting or ending. If a result of 0 is returned at any other time, contact your local SHARP APL representative.

□*AVAIL* requires a dummy right argument of ⍳0 to allow for future extension.

'file-id [size-limit]' □*CREATE* file-number

Establishes a new file name and share-ties the file. The library number may designate either a public library or the user's private library. The file name must differ from any others in that library. The right argument designates an available file number. The optional reservation may designate an amount of file storage different from the default of one hundred thousand bytes.

FILE ACCESS ERROR - User is not authorized to own files or the library number belongs to some other user.

FILE NAME ERROR - Left argument duplicates an existing file id.

FILE QUOTA USED UP - User already owns as many files as his file quota permits.

FILE RESERVATION ERROR - The file size limit designated in the left argument exceeds the user's file space limit.

FILE TIE ERROR - File number is already tied to another file.

FILE TIE QUOTA USED UP - User already has the maximum allowed number of files concurrently tied, (i.e., 20).

□*DROP* file-number, int [,passno]

Drops components from either end of a file. int indicates the number of components to drop from the low-numbered end of the file (if int is positive) or the high-numbered end of the file (if int is negative). If int is zero, no components are dropped. Dropping components does not cause the component numbers of remaining components to be changed.

FILE ACCESS ERROR - User is not permitted to drop components from this file; or incorrect passnumber.

FILE INDEX ERROR - The number of components to be dropped exceeds the number of components in the file.

FILE TIE ERROR - File number is not tied to a file.

'file-id' □*ERASE* file-number [,passno]

Drops all components of a file, unties the file, and deletes the file name. The left and right arguments must designate the same file.

FILE ACCESS ERROR - User is not permitted to erase this file; incorrect passnumber.

FILE NAME ERROR - Left and right arguments do not designate the same file.

FILE TIE ERROR - File number is not tied to a file.

FILE TIED - File is tied by another user.

□*FHOLD* file-number-passno-matrix

Sets an interlock on each file designated in the right argument, after releasing any interlocks that may have been set previously. Elements of the right argument must be distinct, and designate tied files. The right argument can be a file-number-vector **or** a 2-row matrix, where the first row is a file-number-vector and the second row is a vector of corresponding passnumbers. Interlocks will not be set while any other user has an interlock set on any of the designated files, and □*FHOLD* execution will wait until all such other interlocks have been released. A □*FHOLD* only delays execution of other user's □*FHOLD*s. A □*FHOLD* execution delay can be broken artificially by signalling attention several times.

FILE ACCESS ERROR - User is not permitted to □*FHOLD* one or more of the designated files, or the passnumber is incorrect for □*FHOLD*.

FILE TIE ERROR - Some file number appears more than once in the argument; or some element of the argument is not tied to a file.

□*HOLD* file-number-vector

Sets an interlock on each file designated in the right argument, after releasing any interlocks that may have been set previously. Elements of the right argument must be distinct, and designate tied files. Interlocks will not be set while any other user has an interlock set on any of the designated files, and □*HOLD* execution will wait until all such other interlocks have been released. A □*HOLD* only delays execution of other user's □*HOLD*s. A □*HOLD* execution delay can be broken artificially by signalling attention several times.

FILE ACCESS ERROR - User is not permitted to □*HOLD* one or more of the designated files.

FILE TIE ERROR - Some file number appears more than once in the argument; or some element of the argument is not tied to a file.

result ← □*LIB* library-number

Returns a character matrix holding, in each row, the file id of a file in the designated library to which this user is permitted some form of access. The result has 22 columns: columns 1 through 10 contain the library number right-justified, column 11 is blank, and columns 12 through 22 contain the file name left-justified with trailing blanks.

result ← □*NAMES*

Returns a character matrix holding, in each row, the file id of a tied file. The file id's are arranged in the order in which they were tied. The result has the same structure as the result of □*LIB*.

result ← □*NUMS*

Returns a numeric vector holding the file numbers of all tied files. Each file number is associated with the file id in the corresponding row of the result of □*NAMES*.

result ← □*RDAC* file-number [,passno]

Reads the access matrix of a file, and returns its value as the result.

FILE ACCESS ERROR - User not permitted to read access matrix for this file; or incorrect passnumber.

FILE TIE ERROR - File number is not tied to a file.

result ← □*RDCI* file-number, component-number [,passno]

Returns a vector giving certain characteristics of a component. The result is a three-element vector containing:

The workspace storage needed to hold the component, in bytes;

The account number of the person who most recently executed a □*APPEND*, □*APPENDR* or □*REPLACE* on the component;

The time of the most recent □*APPEND*, □*APPENDR* or □*REPLACE* on the component, in 60ths of seconds since midnight, March 1, 1960.

Future extensions to the file system may require that □*RDCI* return a vector of more than three elements. Don't assume that the result of □*RDCI* is always a three-element vector.

FILE ACCESS ERROR - User not permitted to read component information; or incorrect passnumber.

FILE INDEX ERROR - Component number lies outside the component number limits.

FILE TIE ERROR - File number not tied to a file.

result ← □*READ* file-number, component-number [,passno]

Reads a component of a file, and returns its value as the result.

FILE ACCESS ERROR	-	User not permitted to read this file; or incorrect passnumber.
FILE INDEX ERROR	-	Component number lies outside the component number limits.
FILE TIE ERROR	-	File number not tied to a file.

'file-id [size-limit]' □*RENAME* file-number [,passno]

Sets the library number, name, and storage reservation of the file designated by the right argument, as specified by the left argument. The user renaming the file becomes the file owner. The rules governing the left argument are the same as for □*CREATE*, except that the file id may be identical to the subject file's previous file id. If the optional reservation does not appear in the left argument the storage reservation is unchanged.

FILE ACCESS ERROR	-	User not permitted to rename this file; user not authorized to own files; library number belongs to another user; or incorrect passnumber.
FILE FULL	-	Size limit is less than the file size.
FILE NAME ERROR	-	Left argument duplicates an existing file id **other** than that of the subject file.
FILE QUOTA USED UP	-	User already owns as many files as his file quota permits.
FILE RESERVATION ERROR	-	Size limit exceeds the user's remaining file space quota.
FILE TIE ERROR	-	File number is not tied to a file.

value □*REPLACE* file-number, component-number [,passno]

Stores the value of the left argument in an existing component. Appendix C discusses effects of a system failure during execution of □*REPLACE*.

FILE ACCESS ERROR	- User not permitted to replace components of this file; or incorrect passnumber.
FILE FULL	- The new component was not replaced because the file already consumed or exceeded its size limit.
FILE INDEX ERROR	- Component number lies outside component number limits.
FILE TIE ERROR	- File number not tied to a file.

size-limit □*RESIZE* file-number [,passno]

Sets the size limit of the file designated by the right argument to that indicated by the left argument.

FILE ACCESS ERROR	- User not permitted to resize this file; or incorrect passnumber.
FILE FULL	- Size limit is less than the file size.
FILE RESERVATION ERROR	- Size limit exceeds the user's remaining file space quota.
FILE TIE ERROR	- File number is not tied to a file.

result ← □*SIZE* file-number [,passno]

Returns a vector giving certain characteristics of a file. □*SIZE* returns a four-element numeric vector containing:

The number of the first component in the file;

One more than the number of the last component in the file;

The physical storage allocated to the file, in bytes (i.e., the file size);

The physical storage reserved for the file, including that already allocated, in bytes.

Appendix B discusses considerations affecting the third and fourth elements of the result. Future extensions to the file system may require that □*SIZE* return a vector of more than four elements. Don't assume that the result of □*SIZE* is always a four-element vector.

FILE ACCESS ERROR	- User is not permitted to execute □*SIZE*; or incorrect passnumber.
FILE TIE ERROR	- File number not tied to a file.

value □*STAC* file-number [,passno]

Sets the access matrix for a file to the value of the left argument. The left argument must be a 3-column integer matrix with less than 1000 rows; the second columnn of the matrix must be composed of combinations of permission-type values. New access settings are imposed immediately on any users who have the file tied at the time the □*STAC* is executed. Appendix C discusses effects of a system failure during execution of □*STAC*.

FILE ACCESS ERROR - User is not permitted to set the access matrix for this file; or incorrect passnumber.

FILE TIE ERROR - File number is not tied to a file.

'file-id' □*STIE* file-number [,passno]

Establishes a file tie for shared use. The arguments are as for □*TIE*.

FILE ACCESS ERROR - User is not permitted any form of access to this file; incorrect passnumber; or access matrix is damaged. (See Appendix C)

FILE NAME ERROR - File name not found in library.

FILE TIE ERROR - File number is already tied to another file.

FILE TIE QUOTA USED UP - User already has the maximum allowed number of concurrently-tied files, (i.e., 20).

FILE TIED - User already has this file tied; or some other user has the file exclusive-tied.

'file-id' □*TIE* file-number [,passno]

Establishes a file tie for exclusive (non-shared) use. The left argument designates an existing file. The right argument designates an available file number.

FILE ACCESS ERROR - User is not permitted to exclusive-tie this file; incorrect passnumber; or access matrix is damaged. (See Appendix C)

FILE NAME ERROR - File name not found in library.

FILE TIE ERROR - File number is already tied to another file.

FILE TIE QUOTA USED UP - User already has the maximum allowed number of concurrently-tied files, (i.e., 20).

FILE TIED - User already has this file tied; or some other user has the file tied.

□*UNTIE* file-number-vector

Unties all files designated by the right argument. Elements of the right argument must be distinct, and designate tied files. An empty vector is permitted as an argument, and does not affect any file ties.

FILE TIE ERROR - Some file number appears more than once in the argument; or some element of the argument is not tied to a file.

Appendix B

PHYSICAL STORAGE CONSIDERATIONS

The characteristics of the storage devices used to hold files, and the storage allocation strategies used by the file system, make the relationship between component size and file storage space rather complicated. On the average, the number of bytes of file space required to hold an APL value that occupies *N* bytes of workspace storage is:

```
1.04 × (20+N)
```

File storage amounts are always multiples of the physical record size (3156 bytes) used by the file system.

The total storage occupied by a file may be as much as several thousand bytes more than this formula would indicate. As components are appended to a file, the storage used grows unpredictably; appending an empty vector may add several hundred or several thousand bytes, while a subsequent appending of a hundred-element vector may appear to require no additional storage. The reason is that when a file has occupied all allocated storage, appending even a small component causes the allocation of at least an entire physical record.

When components are added by appending, several small components may share a single physical record. When a *□REPLACE* function increases the size of a component, however, the additional physical records that may be needed are not shared with any other component. The space so allocated may be several thousand bytes in excess of what is needed for the component's new value. Additional *□REPLACE*s of that component will use that excess space before any further records are allocated.

Suppose for example, that file number 33 has been built by appending 50 ten-element integer vectors. The space occupied by the file is 9468 bytes. Now, if each component is **replaced** by an eleven-element integer vector, the occupied space becomes 167268 bytes because a new physical record has been allocated for each component. However, replacing each component again by a somewhat larger vector causes **no** increase in file storage. In fact, file storage will not increase again unless the components grow to integer vectors of more than 793 elements.

Storage for the file in this example will **not** decrease if components are now reduced again to ten-element vectors—the additional records that were allocated remain allocated. Dropping a component from the file will, of course, reclaim all records allocated to that component.

There is an important exception to the preceding: routine maintenance procedures that are performed periodically by the computer operations staff may cause all allocated but unused space to be reclaimed. While this action may decrease the space occupied by a file, it also means that the next *□REPLACE* that increases component size, over the size at the time of the maintenance procedure, will once again cause an increase in allocated space.

Appendix C

EFFECT OF SYSTEM FAILURE ON FILE FUNCTIONS

As a general rule, a file function once initiated is carried to a successful conclusion regardless of system failure or user action. In particular, no terminal or communication troubles will prevent a file function from executing successfully. There is, however, the very slight risk that a major system failure may actually cut off a data-transfer function from a workspace to a file. This hazard does not exist on *□APPEND* functions, since a new component's existence is not recorded until the file system has finished any necessary data transfer. Thus a component will be appended correctly or not at all.

There is a hazard that while a *□REPLACE* is executing, a system failure may cut short the data transfer and leave the subject component holding a mixture of old and new values. If this happens, the component cannot be read; executing *□READ* produces a *FILE COMPONENT DAMAGED* report. Thus, damage will never go undetected.

The only functions that can be performed on a damaged component are *□RDCI*, *□REPLACE*, and *□DROP* (or dropping as part of a *□ERASE*). *□RDCI* will return the component information for the latest *□REPLACE*. The damage can be repaired by giving the damaged component a new value through a *□REPLACE*; and of course dropping the component will also eliminate the damage.

In some cases, the computer operations staff can help reconstruct the value of the component. For example, reconstruction is possible if the component is a vector that grows by catenating new elements without changing any elements that were in the previous value. In many cases the old or new value of the component can be obtained from other sources. If the expense of recovery from a damaged-component condition is judged to be severe, it is always possible for the file user to add sufficient safeguards at the cost of executing some additional file functions.

The *□STAC* function is very similar to *□REPLACE*, and incurs the same hazard. If a system failure interrupts a *□STAC*, damaging the access matrix, no one but the file owner will be permitted access to the file until the access matrix has again been set successfully. Storing the new value of the access matrix in another file component before executing *□STAC* provides an adequate safeguard against damage, since the access matrix can later be read from the file and the *□STAC* re-executed.

INDEX

Note on alphabetization: The right parenthesis) is alphabetized as though it didn't exist; thus, *)CONTINUE* is under **C**. "□" is alphabetized as though it were spelt "quad".

A

access control, 2 11
  effect on file's owner, 11
  effect on □*LIB*, □*NAMES*, □*NUMS*, □*TIE*, and □*UNTIE*, 11
  effect on □*STIE*, 11 13
  failed accesses, 11 17
  logging of accesses, 18
  programming precautions, 19
  under program control, 18
  use of "superfluous" parentheses for, 19
  using passnumber: see *passnumber*
  using □*RDAC* and □*STAC*, 11
  see also *access matrix*
access matrix, 13
  account number, 11
  definition, 11
  effect of system failure, 34
  example, 17
  initial shape, 11
  manipulating: see □*RDAC* and □*STAC*
  passnumber: see *passnumber*
  permission: see *permission*
  reading: see □*RDAC*
  setting: see □*STAC*
  see also *access control*
account information, 19
adding a new component (to a file): see □*APPEND* and □*APPENDR*
active files: see *tied files*
appending a component (to a file): see □*APPEND* and □*APPENDR*
applications, shared-file, 17
argument (to file system function), file id as, 3
  noninteger: see *DOMAIN ERROR*
  too many elements: see *LENGTH ERROR*
  wrong arrangement: see *SYNTAX ERROR*
  wrong rank: see *RANK ERROR*
asynchronous processing of files, 13
ATTN or BREAK, 19 22
audit trail, use of □*RDCI* for, 12
availability of file system: see □*AVAIL*

B

backup interrupted: see *FILE BACKUP INTERRUPT*
BREAK or ATTN, 19 22

C

CDATA (example file), 9
changing a file's access matrix: see □*STAC*
changing a file's name: see □*RENAME*
changing a file's size: see □*RESIZE*
communication, interruption of: see *sign-off*
component, appending (to a file): see □*APPEND* and □*APPENDR*
  compared with variable in workspace, 2
  damaged: see *FILE COMPONENT DAMAGED*
  date and time appended or last replaced, 12
  definition, 2
  destroyed: see □*ERASE*
  dropping: see □*DROP*
  first used: see □*SIZE*
  how many in file: see □*SIZE*
  information: see *component information*
  number, next available: see □*SIZE*
  reading: see □*READ*
  released in an arbitrary sequence, 9
  replacing: see □*REPLACE*
  size limit, 2
  user that appended or last replaced, 12
  workspace storage required, 12
component information, 12
  reading: see □*RDCI*
component number, calculating the value of, 6
  definition, 2
  example, 3
  next available: see □*SIZE*
)CONTINUE (system command): see *sign-off*
CONTINUE (workspace), 5
controlling access to files: see *access control*
conversation, terminal-to-terminal, 15
CONVERSE (example program), 15
creating a file: see □*CREATE*

D

damaged component or file: see *FILE COMPONENT DAMAGED*, *FILE DAMAGED*
DEBIT (example variable), 7
decision-making exercises, strategic, 15
deleting a component: see □*DROP*
deleting a file: see □*ERASE*
displaying file size: see □*SIZE*
DOMAIN ERROR (error report), 22
dropping a component: see □*DROP*
dropping a file: see □*ERASE*

E

erasing a component: see □*DROP*
erasing a file: see □*ERASE*
error reports, 3 22 23
  summary table by file function, 24
errors, file, 3

F

failure, of power: see *sign-off*
  of system, 34
FIFO (first in, first out) file, implemented with □*DROP* and □*SIZE*, 8
file, access to: see *access control*
  adding to: see □*APPEND* and □*APPENDR*
  asynchronous processing, 13
  compared with workspace, 1 2 3
  creating: see □*CREATE*
  damaged: see *FILE DAMAGED*
  definition, 2
  erasing: see □*ERASE*
  errors, 3
  identification of: see *file id*
  identification of, changing: see □*RENAME*
  interlocking: see □*FHOLD* and □*HOLD*
  merging with another (example), 6
  moving: see □*RENAME*
  name: see *file name*
  number tied (i.e., active), 2
  ownership, 2
  quota: see *file quota*, *file size quota*, and *file tie quota*
  reading: see □*READ*
  renaming: see □*RENAME*
  residence, 2
  resizing: see □*RESIZE*
  security: see *access control*
  sharing, 1 11
  size: see *file size*
  size limit, 2
  size quota: see *file size quota*
  storage requirements, 7 33
  tie quota: see file tie quota
  tying: see □*STIE* and □*TIE*
  untying: see □*UNTIE*
  updating: see □*APPEND*, □*APPENDR*, □*DROP*, and □*REPLACE*
  usage, 2
  writing to: see □*APPEND*, □*APPENDR*, and □*REPLACE*
FILE ACCESS ERROR (error report), 11 17 24 25 26 27 28 29 30 31
FILE BACKUP INTERRUPT (error report), 22
FILE COMPONENT DAMAGED (error report), 22 34
FILE DAMAGED (error report), 22
FILE FULL (error report), 24 25 29 30
file functions, definition, 2
  summary, 20
  see also □*APPEND*, □*APPENDR*, □*AVAIL*, □*CREATE*, □*DROP*, □*ERASE*, □*FHOLD*, □*HOLD*, □*LIB*, □*NAMES*, □*NUMS*, □*RDAC*, □*RDCI*, □*READ*, □*RENAME*, □*REPLACE*, □*RESIZE*, □*SIZE*, □*STAC*, □*STIE*, □*TIE*, and □*UNTIE*
file id, changing : see □*RENAME*
  definition, 3
  improper: see *DOMAIN ERROR*
FILE INDEX ERROR (error report), 24 25 26 28 29 30
FILE INTERRUPT (error report), 22
FILE LIBRARY NOT AVAILABLE (error report), 22
file name, 3
  see also *file id*
FILE NAME ERROR (error report), 24 26 27 29 31
file number, 3
  calculating the value of, 6
file quota, 23
  see also *file space quota* and *file tie quota*
FILE QUOTA USED UP (error report), 23 24 26 29
file quotas, 2
FILE RESERVATION ERROR (error report), 23 24 26 29
file sharing, concurrent, 13
file size, 8
  displaying: see □*SIZE*
file size limit, 8
  changing: see □*RESIZE*

file space quota, 23
used up: see *FILE RESERVATION ERROR*
file system, displaying availability of: see *□AVAIL*
FILE SYSTEM ERROR (error report), 22
FILE SYSTEM HARDWARE ERROR (error report), 22
FILE SYSTEM NO SPACE (error report), 22
FILE SYSTEM NOT AVAILABLE (error report), 22
FILE SYSTEM TIES USED UP (error report), 23
file tie, 3
listing: see *□NAMES* and *□NUMS*
maintenance, 5
none available: see *FILE SYSTEM TIES USED UP*
not changed by system commands, 6
FILE TIE ERROR (error report), 24 25 26 27 28 29 30 31 32
file tie number: see *file number*
file tie quota, 23
FILE TIE QUOTA USED UP (error report), 23 24 26 31
FILE TIED (error report), 13 24 27 31
FILEAID (workspace), for *□RDCI*, 12
files, tied, listing: see *□NAMES* and *□NUMS*
first in, first out (FIFO) file, implemented with *□DROP* and *□SIZE*, 8
function, file: see *file functions*

H

holding file processing: see *□FHOLD* and *□HOLD*

I

identification of a file: see *file id*
immediate-execution mode, effect on *□FHOLD*, 14
interleaved processing, 13
interlock of file processing: see *□FHOLD* and *□HOLD*
interrupt, of communication: see *sign-off*
of file operation: see *FILE INTERRUPT*
of system file backup: see *FILE BACKUP INTERRUPT*

L

last in, first out (LIFO) file, implemented with *□DROP* and *□SIZE*, 8
LENGTH ERROR (error report), 22
library, files in: see *□LIB*
listing: see *□LIB*
unavailable: see *FILE LIBRARY NOT AVAILABLE*
library number, 2
see also *file id*
LIFO (last in, first out) file, implemented with *□DROP* and *□SIZE*, 8
listing a library: see *□LIB*
listing tied files: see *□NAMES* and *□NUMS*
locked functions, use of passnumber in, 17
logging of accesses, by system, 11 17
by user program, 18

M

magic number: see *passnumber*
matrix: see *access matrix*
merging files (example), 6
moving a file: see *□RENAME*
multilateral games, 15
multiprocessing, simulation of, 15

N

next available component: see *□SIZE*
noninteger argument: see *DOMAIN ERROR*

O

)OFF (system command): see *sign-off*
operation interrupted: see *FILE INTERRUPT*

P

passnumber (element of access matrix), 3 11 12
example, 17 19
mismatching, 17
nonzero, use of, 12 16
use as constant only, 19
use in locked functions, 17
permission (element of access matrix), 11
example, 17
negative integer as value, 12
values, 12 20
PERS (example file), 17
PERSONS (example file), 3 4 5 6

physical record size, 33
power failure: see *sign-off*
PREAD (example function), 17
PRINTINVOICE (example function), 9
private versus public files, 2
processing, holding: see □*FHOLD* and □*HOLD*
programs, controlling sequencing between, 15
PROG1 (example program), 4
PROG2 (example program), 5
PROG3 (example program), 6
PROG4 (example program for dropping empty components), 9
PTIE (example program), 17

Q

□*AI*, 19
□*APPEND*, 3 15 28
  definition, 4 25
  effect of system failure, 34
  errors, 24 35
  example, 4 6 8 22
  summary, 20
  use in terminal-to-terminal conversation, 15
□*APPENDR*, 28
  definition, 10 25
  errors, 24 25
  summary, 20
  usage, 15
□*AVAIL*, definition, 10 25
  summary, 20
□*CREATE*, 29
  compared with □*TIE*, 5
  definition, 4 26
  errors, 23 24 26
  example, 4 6
  summary, 20
  syntax, 20
  use in terminal-to-terminal conversation, 15
□*DROP*, definition, 7 8 26
  effect of system failure, 34
  errors, 24 26
  example, 8 9
  summary, 20
  with FIFO and LIFO files, 8
□*ERASE*, 23
  definition, 6 27
  effect of system failure, 34
  errors, 24 27
  example, 6
  summary, 20
□*FHOLD*, definition, 13 27
  errors, 24 27
  example, 14 15
  summary, 20
  usage, 13 14 15 16 19
□*HOLD*, definition, 27
  errors, 24 27
  summary, 20
□*LIB*, definition, 6 28
  effect of access control, 11
  summary, 20
□*NAMES*, definition, 7 28
  effect of access control, 11
  summary, 20
□*NUMS*, definition, 7 28
  effect of access control, 11
  summary, 20
  with □*UNTIE*, 7
□*RDAC*, definition, 12 28
  errors, 24 28
  summary, 20
□*RDCI*, definition, 12 28
  effect of system failure, 34
  errors, 24 28
  summary, 20
  usage, 12 19 34
□*READ*, 3 17
  definition, 5 29
  effect of system failure, 34
  errors, 24 29
  example, 5 6 8 9 13 14 15 16 17 18 19 22
  summary, 20
□*RENAME*, definition, 9 29
  errors, 23 24 29
  summary, 20
  syntax, 20
  used for moving a file (example), 9
□*REPLACE*, 28
  definition, 7 30
  effect of system failure, 34
  effect on storage requirements, 33
  errors, 24 30
  example, 9 13 14 15 19
  summary, 20
□*RESIZE*, definition, 9 30
  errors, 24 23 30
  example, 10
  summary, 20
□*SIZE*, 23
  definition, 8 30
  errors, 24 30
  example, 8 9
  summary, 20

with FIFO and LIFO files, 8
□*STAC*, definition, 11 31
effect of system failure, 34
errors, 24 31
summary, 20
□*STIE*, definition, 13 31
effect of access control, 11 13
errors, 23 24 31
example, 15 17 22
summary, 20
□*TIE*, compared with □*CREATE*, 5
definition, 5 31
effect of access control, 11
errors, 23 24 31
example, 5 9
summary, 20
usage, 13
□*UNTIE*, definition, 4 32
effect of access control, 11
errors, 24 32
example, 4 6 9 15
summary, 20
with □*NUMS*, 7
quota, file: see *file quota*
file space: see *file space quota*
file tie: see *file tie quota*

R

RANK ERROR (error report), 22
reading a component: see □*READ*
reading an access matrix: see □*RDAC*
reading component information: see □*RDCI*
reconstructing files, 34
record size, physical, 33
RECORDS (example file), 6
reidentifying a file: see □*RENAME*
releasing components in an arbitrary sequence (example), 9
renaming a file: see □*RENAME*
replacing a component: see □*REPLACE*
resizing a file: see □*RESIZE*

S

SALES (example file), 3 4 5 6
scratching a file: see □*ERASE*
sequencing between programs, controlling, 15
setting an access matrix: see □*STAC*
shared tie, 13
shared-file applications, 17
sharing a file, 11
sign-off, effect on □*FHOLD*, 14
effect on tied files, 6
simulation of asynchronous but interdependent activities, 15
size, physical record, 33
space quota: see *file space quota*
storage considerations, 7 33
storing an access matrix: see □*STAC*
storing a component: see □*APPEND*, □*APPENDR*, and □*REPLACE*
strategic decision-making exercises, 15
SYNTAX ERROR (error report), 22
system, displaying availability of: see □*AVAIL*
system error: see *FILE SYSTEM ERROR*, *FILE SYSTEM HARDWARE ERROR*, and *FILE SYSTEM NO SPACE*
system failure, 34
system unavailable: see *FILE SYSTEM NOT AVAILABLE*

T

TALKTALK (example file), 15
telephone interruption: see *sign-off*
terminal-to-terminal conversation, 15
tie, file: see *file tie*
tie number: see *file number*
tie quota: see *file tie quota*
tied files, listing names: see □*NAMES*
listing file numbers: see □*NUMS*
tying a file: see □*CREATE*, □*STIE*, and □*TIE*
tying files, definition, 3
TIME (public function), for □*RDCI*, 12
TIMEN (public function), for □*RDCI*, 12

U

untying files: see □*UNTIE*
updating a component: see □*REPLACE*

V

variable (in workspace), compared with component, 2

W

workspace, compared with file, 1 2
full: see *WS FULL*
writing to a file: see □*APPEND*, □*APPENDR*, and □*REPLACE*
WS FULL (error report), 22

Reader's Comments

You can help us improve future editions of this manual by taking the time to send us your comments. We appreciate feedback on our publications, and will gladly reply to your comments, if you wish.

Some topics for comment are: readability, use of examples, organization, clarity, suggested improvements.

CUT ALONG THIS LINE

Thank you for your comments. Would you like a reply? _____

Name _____

Company _____

Address _____

_____

_____

Please return this form to your nearest I.P. Sharp office or:

Publications Department
I.P. Sharp Associates Ltd.
145 King Street West
Toronto, Ontario
Canada M5H 1J8

ISBN 0 86493 045 3